Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Blah abstrakt…

1 EINLEITUNG

Mit E-Commerce-Beispiel motivieren

2 GRUNDLAGEN VON ENTERPRISE-ARCHITEKTUREN

Verteilte Systeme …
  Architekturen …
  Komponenten …
  …

3 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

Blah …

4 MODERNE ENTERPRISE-ARCHITEKTUREN

4.1 Microkernel Architecture

Die Microkernel Architecture ist ein Architekturmuster, was sich durch Erweiterbarkeit, Flexibilität und vor allem Isolation der Funktionalitäten auszeichnet. Wie in Abbildung 1 dargestellt, enthält ein Microkernel zwei wesentliche Komponenten: Den Kern der Anwendung,der die wichtigsten grundlegenden Funktionalitäten bereitstellt und Module oder auch Plugins, die diesen Kern um Features erweitern [Richards 2015, 21-22].

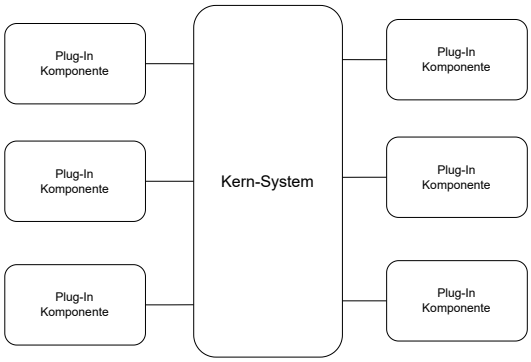

Abb. 1. Aufbau einer Microkernel Architecture

Der Kern der Anwendung implementiert dabei meist nur die minimalste Funktionalität, um die Anwendung oder das System lauffähig zu machen. Alle weiteren Funktionalitäten werden in Modulen implementiert, die auf den Kern aufbauen. Module sind meist unabhängig voneinander aufgebaut, es kann jedoch auch vorkommen, dass manche Module von anderen abhängig sind. Best practice für die Entwicklung von Microkernel Architekturen ist es, die Kommunikation zwischen einzelnen Modulen so gering wie möglich zu halten, um Probleme durch Abhängigkeiten zu vermeiden. Dadurch sind Module untereinander lose gekoppelt und können unabhängig voneinander entwickelt, getestet und deployed werden [Richards 2015, 22].

Die Plugins können über verschiedene Wege mit dem Kern verbunden werden. Eine genaue Spezifikation zum Verbinden der Plugins mit dem Kern gibt es aber laut Architekturschema nicht, diese Entscheidung ist dem Entwickler überlassen und entsprechend der Anforderungen und Anwendungsumgebung zu treffen. Unabhängig von der Art der Verbindung definiert der Kern die Schnittstellen, um Plugins anzubinden. Diese Verbindung könnte dann beispielsweise über Web-Services, Messaging oder am einfachsten über direkte Objekt-Instanziierung innerhalb der gleichen Anwendung stattfinden [Richards 2015, 22-23].

Zwischen Plugins und Kern werden Verträge definiert, die die Kommunikation zwischen den beiden Komponenten regeln. Diese Verträge können in Form von Interfaces, Klassen oder auch Datenstrukturen definiert werden. Alle Plugins, müssen sich zwingend an die definierten Verträge halten, um mit dem Kern kommunizieren zu können. Alternativ können auch Adapter verwendet werden, um bestehende Plugins an den Kern und die Verträge anzupassen, wodurch wiederrum die lose Kopplung der Komponenten verbessert wird.

Durch diesen Aufbau ergibt sich jedoch das Problem, dass der Kern jederzeit über Verfügbarkeit und Erreichbarkeit der Plugins informiert sein muss. Um dieses Problem zu lösen, kann eine zentrale Plugin-Registry verwendet werden. Diese Registry enthält alle aktuell verfügbaren Plugins sowie die dazugehörigen relevanten Informationen wie zum Beispiel Name des Service, Verträge, Verbindungsdetails, etc. Der Kern der Anwendung kann dann zur Laufzeit auf diese Registry zugreifen und Plugins dynamisch laden [Richards 2015, 22].

Microkernel Architekturen können auch in andere Architekturmuster eingebettet werden, falls es nicht möglich sein sollte die gesamte Software in diesem Architekturmuster aufzubauen. Vor allem Teile von Anwendungen, die stark erweiterbar sein müssen, eignen sich gut für die Verwendung der Microkernel Architektur.

Ein klarer Vorteil dieses Architekturmusters ist die schnelle Reaktionsfähigkeit auf äußere Änderungen, da Anpassungen aufgrund der losen Kopplung größtenteils nur in den isolierten Modulen vorgenommen werden. Der Kern der Anwendung ist in den meisten Fällen schnell stabil und benötigt selten im Laufe der Entwicklung weitere Angleichungen. Geänderte Module können je nach Implementierung auch zur Laufzeit geladen oder hinzugefügt werden,

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

was mögliche Downtime von bereits ausgelieferter Software minimiert [Richards 2015, 25].

Ein Beispiel dafür stellt die Entwicklung von Betriebssystem-Kernels dar, die auch namensgebend für dieses Architekturmuster ist. Deren Kern Komponenten sind in der Regel sehr stabil und implementieren vor allem grundlegende Funktionen wie Speicherverwaltung, Prozessverwaltung und I/O-Operationen. Weitere low-level Funktionalitäten wie beispielsweise Geräte Treiber oder Dateisysteme werden als Module in den Kernel geladen und können bei Bedarf hinzugefügt oder entfernt werden, was vor allem für die Unterstützung neuer Hardware wichtig ist.

Auch Entwicklungsumgebungen in der Softwareentwicklung nutzen oft Microkernel Architekturen, um die Support für verschiedene Programmiersprachen und Frameworks zu ermöglichen.

Zwar bietet das Beispiel der E-Commerce Anwendung keinen klassischen Anwendungsfall für Microkernel Architekturen, jedoch kann die Verwendung von Microkernel Architekturen in Teilen der Anwendung trotzdem sinnvoll sein. Sowohl Zahlungs- als auch Versandfunktionalitäten könnten, wie in Abbildung 2 dargestellt, in Module ausgelagert werden, um die Anwendung durch weitere Dienstleister erweitern zu können.

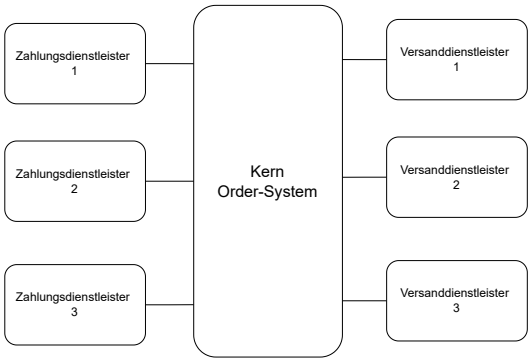

Abb. 2. Aufbau der E-Commerce Anwendung mit Microkernel Architektur

Logik zum Verarbeiten der Zahlungen und Versandinformationen wird dann in den Modulen implementiert, die auf den Kern der Anwendung aufbauen. Dabei ist jedoch ein Großteil der Business-Logik im Kern der Anwendung enthalten, was nicht dem eigentlichen Gedanken der Microkernel Architektur entspricht.

Abgesehen davon erhöhen Microkernel Architekturen inhärent die Testbarkeit der Software, da Module nur lose Kopplung gekoppelt sind. Jedes Modul kann unabhängig voneinander getestet werden und fehlende Module durch Stubs ersetzt werden, wodurch sich während der Entwicklung auf einzelne Module isoliert konzentriert werden kann [Richards 2015, 26]. Weiterhin können Verhaltensweisen von anderen Modulen durch Mocks simuliert werden, um Testzustände zu erzeugen und das Verhalten der Anwendung zu verifizieren. In Agilen Umgebungen, in denen das Testen von Software eine wichtige Rolle spielt, ist die Verwendung von Microkernel Architekturen daher besonders sinnvoll.

Eine Herausforderung bei der Verwendung von Microkernel Architekturen kann jedoch der Entwurf der Kern-Komponente darstellen. Da alle anderen Module auf den Kern aufbauen, muss dieser sehr sorgfältig und stabil entwickelt werden, um die Funktionalität der gesamten Anwendung zu gewährleisten. Diese Rolle sollten vor allem erfahrene Entwickler übernehmen, da sich Design-Fehler der Kern-Komponente oder Verträge negativ auf die Entwicklung der Module auswirken können. Sollte der Kern der Anwendung angepasst werden, so müssen tendenziell auch alle Module überprüft oder aktualisiert werden, was zu erheblich erhöhten Entwicklungszeiten führen kann und zuvor gewonnene Vorteile der Microkernel Architektur zunichte macht.

Aufgrund der initial hohen Komplexität, die mit der Entwicklung des Kerns einhergeht, stellt die Mikrokernel Architektur nicht die beste Wahl dar, wenn es darum geht schnell eine erste Version der Software auf den Markt zu bekommen. Die wesentlichen Vorteile, die die Microkernel Architektur bietet, zeigen sich erst im späteren Verlauf der Entwicklung, wenn die Anwendung erweitert werden muss. Sowohl Iterationen als auch Auslieferungszeiten sind dann sehr kurz, da Module unabhängig voneinander entwickelt werden und die Anwendung schnell an neue Anforderungen angepasst werden kann. Diese Vorteile können die Architektur in ausgewählten Anwendungsfällen sehr geeignet für agile Entwicklungsumgebungen machen.

4.2 Microservice-Architecture

4.3 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier *„An introduction to Software Architecture“* auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Komponente, die auf publizierte Events reagiert

- Vermittler (englisch *Mediator*): Komponente zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 3 stellt diesen Vertrag dar.

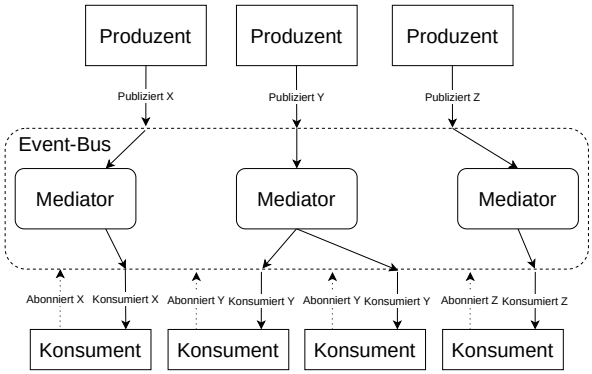

Abb. 3. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht kleinen und autonomen Entwickler-Teams die klare Abgrenzung von Features und folglich einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung hohe Skalierung und die Möglichkeit für Echtzeit-Software. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Events:

- `OrderCreated`: Ein Event, das genau dann erzeugt wird, wenn eine neue Bestellung aufgegeben wird
- `PaymentProcessed`: Ein Event, das genau dann erzeugt wird, wenn der Bezahlvorgang abgeschlossen wird
- `ShipmentInitiated`: Ein Event, das genau dann erzeugt wird, wenn die Bestellung versandt wird

Analog zur Microservice-Architektur teilen wir die Funktionalitäten in drei verschiedene Dienste auf: `OrderService`, `PaymentService` und `ShipmentService`.

Wie Abbildung 4 zeigt, sind alle drei Dienste Produzenten und Publisher, erzeugen also Events und veröffentlichen diese. Die Dienste

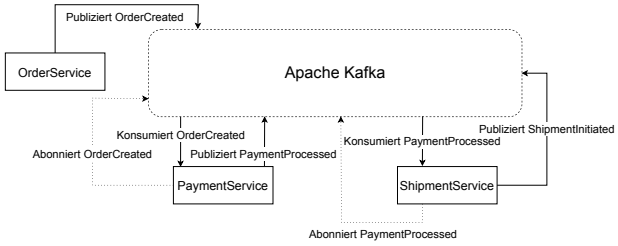

Abb. 4. E-Commerce-Beispiel mit Event-Driven Architecture

`PaymentService` und `ShipmentService` sind zudem Konsumenten, sodass ersterer auf Events des Typs `OrderCreated` und zweiterer auf Events des Typs `ShipmentInitiated` reagiert. Eine beispielhafte Implementierung des `PaymentService` mit Apache Kafka als Event-Broker ist im Anhang A.1 zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [1] zu finden.

Das Beispiel zeigt, dass die Event-Driven Architektur mit weiteren agilen Strukturen wie Microservices kombiniert werden kann, was die Agilität der Architektur weiter erhöht. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

4.4 Cloud-Native Architecture

Die Cloud-Native Architecture beruht auf der Annahme, dass die Infrastruktur in ständigem Wandel ist und die Auslagerung jener in die Cloud mehr Agilität schafft [Gannon et al. 2017]. Als Cloud wird in diesem Fall die Infrastruktur eines oder mehrerer Cloud-Vendors bezeichnet. Ein Cloud-Vendor bietet seine Infrastruktur und deren Verwaltung dabei gegen eine Gebühr an. Unter anderem offeriert er:

- Die globale Nutzung von Ressourcen durch Geo-Redundanz und somit starke Verteilung sowie hohe Verfügbarkeit von Software,
- dynamische Skalierung bereitgestellter Ressourcen basierend auf der Nachfrage von Software (Auto-Scaling),
- nutzungsbedingte Gebühren, sodass nur tatsächlich verwendete Ressourcen bezahlt werden (Pay-as-you-go),
- unterbrechungsfreie Updates von Software (Zero Downtime).

Als cloud-nativ wird hierbei all jene Software bezeichnet, welche explizit für die Cloud entwickelt wurde. Grundsätzlich baut die cloud-native Architektur auf die in diesem Artikel bereits in Abschnitt 4.2 erklärte Microservice-Architektur und die in Abschnitt 4.3 erklärte Event-Driven Architecture auf. Zusätzlich kommen sogenannte *Fully Managed Cloud-Services* hinzu, was cloud-spezifische

[1] https://github.com/Beleg-6-EAP/demo-eda-ecommerce

und vom Cloud-Vendor vollständig verwaltete Dienste sind. Jene umfassen alle von Datenbanken über Message-Queues bis hin zu Serverless-Functions und viele mehr. Charakterisiert werden diese durch die Eigenschaft, dass sich der Entwickler gänzlich auf die Business-Logik konzentrieren kann, da der Cloud-Provider die vollständige Verwaltung der Infrastruktur übernimmt. Zentral dabei ist der Aspekt der Containerisierung, bei welchem jede Komponente eines Systems in einen Container gepackt wird. Der Cloud-Vendor kann die Gesamtheit der Container dann dynamisch orchestrieren und somit gezielt den Ressourcenverbrauch optimieren.

Im Folgenden greifen wir erneut das Beispiel E-Commerce aus der Einleitung auf und betrachten die Anpassung der in Abschnitt 4.3 angewendeten Event-Driven-Architecture auf die Cloud-Native Architecture mit Cloud-Vendor AWS.

Dabei ersetzen wir den dort verwendeten Broker durch eine *EventBridge* [2] - ein serverless Cloud-Service von AWS zum Routen von Ereignissen. Weiter werden die drei Services für Aufgeben einer Bestellung, Bearbeitung der Bezahlung und Initiierung des Versands als *Lambda* [3], also Serverless-Function implementiert. Abbildung 5 stellt diese cloud-native Architektur dar.

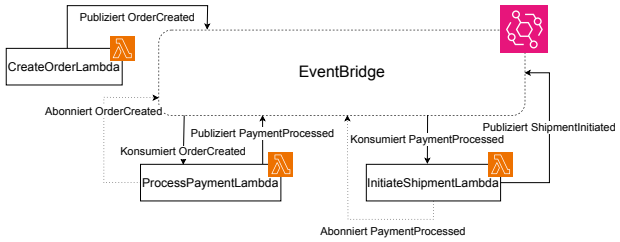

Abb. 5. E-Commerce-Beispiel mit Cloud-Native Architecture

Damit kombiniert die Cloud-Native Architecture die agilen Vorteile der Microservice- und Event-Driven Architecture. Weiter ermöglichen die Cloud-Vendors nahtlose und einfache Möglichkeiten zur Auslieferung der Software, wie beispielsweise Code-Deploy [4] bei AWS. Die Option, allen Fokus von der Infrastruktur auf die Entwicklung zu legen, ermöglicht gemeinsam mit der einfachen Auslieferung kurze Iterationen in agilen Teams und somit noch kürzere Zyklen in der Bereitstellung von Software. Dabei ist besonders die Time-to-Market sehr gering, sodass Tech-Start-Ups häufig cloud-native Architekturen für ihre Software wählen.

Nicht zu vergessen ist zudem die extrem hohe Flexibilität der eingesetzten Ressourcen durch das Auto-Scaling. Das damit verbundene Pay-as-you-go ermöglicht weiter auch finanzielle Agilität, wodurch zum Beispiel keine Vorab-Investitionen für Infrastruktur notwendig sind. Jedoch ist zu betonen, dass aufgrund dessen auch Kostenrisiken durch möglicherweise unerwartete, extrem hohe Nachfrage der Software bestehen. Ebenfalls nicht zu vernachlässigen ist die enge Bindung an den Cloud-Vendor durch Verwendung seiner spezifischen Cloud-Dienste. Im Fall einer Migration zu einem anderen Cloud-Vendor könnten sowohl hohe Kosten beim vorherigen Cloud-Vendor, als auch Entwicklungskosten durch die notwendige Anpassung der genutzten spezifischen Cloud-Dienste anfallen. Obwohl das die Wahl des Cloud-Anbieters und der damit verbunden Entwicklung weniger agil macht, gilt die Cloud-Native Architecture besonders aufgrund der Auslagerung der Infrastruktur als eine - und vermutlich die agilste Architektur für moderne Software.

[2]https://aws.amazon.com/de/eventbridge/, abgerufen am 06.01.2025
[3]https://aws.amazon.com/de/lambda/, abgerufen am 06.01.2025
[4]https://aws.amazon.com/de/codedeploy/, abgerufen am 06.01.2024

5 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

6 DISKUSSION

7 ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

Dennis Gannon, Roger Barga, and Neel Sundaresan. 2017. Cloud-Native Applications. *IEEE Cloud Computing* 4, 5 (2017), 16–21. https://doi.org/10.1109/MCC.2017.4250939

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015. *Software Architecture Patterns.* O'Reilly Media, Inc. value pages.

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

A CODE-BEISPIELE

A.1 Event-Driven-Architecture

```java
package demo.eda.service;

import demo.eda.event.OrderCreatedEvent;
import demo.eda.event.PaymentProcessedEvent;
import demo.eda.model.Payment;
import demo.eda.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Flux;
import reactor.kafka.receiver.KafkaReceiver;
import reactor.kafka.receiver.ReceiverOptions;

import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;
    private final ReactiveKafkaProducerTemplate<String, PaymentProcessedEvent> producer;

    @KafkaListener(topics = OrderCreatedEvent.TOPIC, groupId = "payment-service")
    public Flux<Void> processPayment(ReceiverOptions<String, OrderCreatedEvent> receiverOptions) {
        return KafkaReceiver.create(receiverOptions)
                .receive()
                .flatMap(record -> {
                    final OrderCreatedEvent event = record.value();
                    final Payment payment = new Payment(UUID.randomUUID().toString(), event.getOrderId(), true);
                    return paymentRepository.save(payment).flatMap(savedPayment -> {
                        final PaymentProcessedEvent paymentEvent =
                            new PaymentProcessedEvent(
                                    savedPayment.getOrderId(),
                                    savedPayment.getId(),
                                    savedPayment.isSuccess());
                        return producer.send(PaymentProcessedEvent.TOPIC, paymentEvent).then();
                    });
                });
    }
}
```

Listing 1. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1 mit Apache Kafka als Event-Broker

B ÜBUNGSAUFGABEN

B.1 Übungsaufgabe 1

Blah …